FAME—A Polyglot Library for Metamodeling at Runtime*

Adrian Kuhn and Toon Verwaest

Software Composition Group,
University of Berne, Switzerland
http://scg.iam.unibe.ch

Abstract. Tomorrow's eternal software system will co-evolve with their context: their metamodels must adapt at runtime to ever-changing external requirements. In this paper we present FAME, a polyglot library that keeps metamodels accessible and adaptable at runtime. Special care is taken to establish causal connection between fame-classes and host-classes. As some host-languages offer limited reflection features only, not all implementations feature the same degree of causal connection. We present and discuss three scenarios: 1) full causal connection, 2) no causal connection, and 3) emulated causal connection. Of which, both Scenario 1 and 3 are suitable to deploy fully metamodel-driven applications.

Keywords: causal connection, eternal systems, metamodeling at runtime.

1 Why Metamodeling at Runtime?

Metamodeling is at the core of many web- and business applications. It is the means through which meta-information about the domain of a system is represented. Metamodeling is traditionally limited to the static (*i.e.* design time) representation of meta-information. Most often, only business data is extendible and editable at runtime, whereas any change to the business model requires a re-design of the system, often involving major engineering effort. For example, an application may allow users to edit the content of forms, but not their structure and workflow.

In the vision of eternal software [1], running systems are imagined to adapt to various, often unanticipated, context changes with little or no engineering effort. An eternal system must co-evolve with its context: as the business changes over time, the system is required to extend and adapt its metamodel. Therefore, we advocate to extend common systems with the ability to not only edit and extend the contained data but also the metamodel at runtime. To meet this requirement, the structures of meta-information represented in a metamodel are to be kept accessible *and adaptable* at runtime. As such, they can be used to change and extend the metamodel at runtime—that is *after* the system has been put in use.

This approach has been realized in a library called FAME, which provides a lightweight kernel to represent both models (*i.e.* data) and metamodels using the same structures

* In Proceedings of the 3th Models@Run.time (MRT '08).

at runtime. FAME has initially been developed as the kernel of Moose [2], a highly-adaptable Smalltalk application used for research in software- and information visualization. Within the last year, the library has been ported (at varying stages of specification conformance) to: Squeak, Java, Python, and partially, C# and Ruby.

This paper presents a complete description of Fame's API and design, including its lightweight meta-metamodel (FM3) and text-based exchange format (MSE). Design decisions and implementation issues are discussed. In particular, as some languages offer limited reflective features only, not all implementations of Fame feature the same degree of causal connection. We present and discuss three scenarios: 1) full causal connection, 2) no causal connection, and 3) emulated causal connection, of which both Scenarios 1 and 3 are suitable to deploy fully metamodel-driven applications.

In general, the approach taken by Fame is related to previous work on runtime metamodeling by Costa et al [3], Jouault et al [4], and Clark et al [5]—please refer to our previous work [6] for comprehensive list of references and related work. Parts of Fame's implementation originate from Renggli's Magritte library [7].

The remainder of the paper is structured as follows. Section 2 starts with an overview of FAME and discuss afterwards the core abstractions: in Section 3 the meta-architecture, in Section 4 the FM3 meta-metamodel, in Section 5 the Fame API, and in Section 6 model serialization. In Section 7 we present and discuss three scenarios for causal connection between fame-classes and host-classes. Finally, Section 8 concludes.

2 Fame in a Nutshell

The purpose of FAME is to attach meta-information to the objects of a running system. Fame provides a uniform interface on both objects and their meta-information, that is, both are manipulated with the same API calls.

Since Fame is a polyglot library we have taken special care to ensure that all implementations offer the same core API. Fame has currently been ported to: Smalltalk, Java, Python, and partially, C# and Ruby. The Java library acts as the reference implementation, whereas the Smalltalk library offers the most additional features.

At the very heart of Fame is a *tower of models* with three layers [8]. Each layer contains elements which conform to the meta-information specified by the layer above. The bottom layer M1 contains objects of your running application, the middle layer M2 contains meta-information about your objects, and the top-most layer M3 contains meta-information about the meta-information of you objects. Thus, we refer to these layers as model, metamodel and meta-metamodel layer. Even though, at runtime, any layer is editable, it is common to populate the top-most layer with a static set of elements. This static set of elements is referred to as FM3, the FAME meta-metamodel.

Additionally, elements of any layer are serializable to text stream and back using the MSE exchange format. MSE is a generic file format to exchange any kind of objects, regardless of the metamodel. Thus, FAME-compliant applications can exchange both data (*i.e.* models) and metamodels by the same means.

The complete specification of both FM3 and MSE, as well as the sources of FAME are available at http://smallwiki.unibe.ch/fame under GPL license.

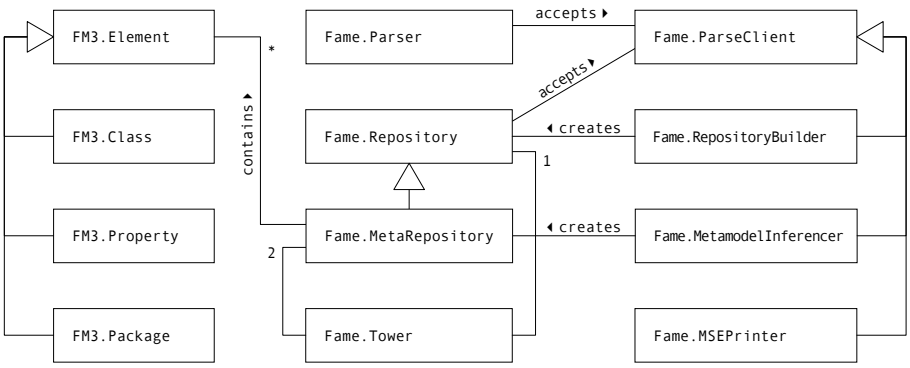

Fig. 1. Class diagram of FAME: including the FM3 meta-metamodel (left), Fame meta-architecture and API (center), and MSE serialization/streaming (right, top center).

Design and Core Abstractions of Fame

All FAME implementations are realized using the same design. Figure 1 presents a class diagram with the main host-classes[1], from left to right: Element, Class, Property and Package implement the FM3 meta-metamodel. Tower, Repository, and MetaRepository implement the meta-architecture and most of Fame's API. Parser, ParseClient and its subclasses are used for serialization and de-serialization of models.

3 Meta-Architecture —– Towers – Models – Elements

The major components of Fame's meta-architecture are towers, models, and elements. Towers contain models, models contain elements. There are two kinds of models: meta-repositories and repositories. The former contain FM3-compliant elements only, whereas the latter my contain any kind of host-language objects, typically domain objects. Often the terms model and repository are used interchangeably.

Each tower consists of three layers referred to as, from bottom to top: M1, M2, and M3. The elements on every layer must conform to the meta-information specified by the layer above. A typical setup is as follows: at the bottom layer a repository holding domain objects of the running application, at the middle layer a meta-repository holding the current domain model, and eventually, at the top-most layer, a meta-repository holding the self-described FM3 metamodel.

The tower is not implemented as a singleton because more than one application may run in the same object memory, each with its own tower. These towers may share the top layers. For example, if each open document of an application is represented by a concurrent tower, then the towers have different M1 layers but share the other layers.

[1] The term *class* is used by both object-oriented programming and the metamodeling paradigm, but with different meanings. Hence, we refer to the classes of the host-language as *host-classes* and to the classes of FM3-compliant metamodels as *fame-classes*.

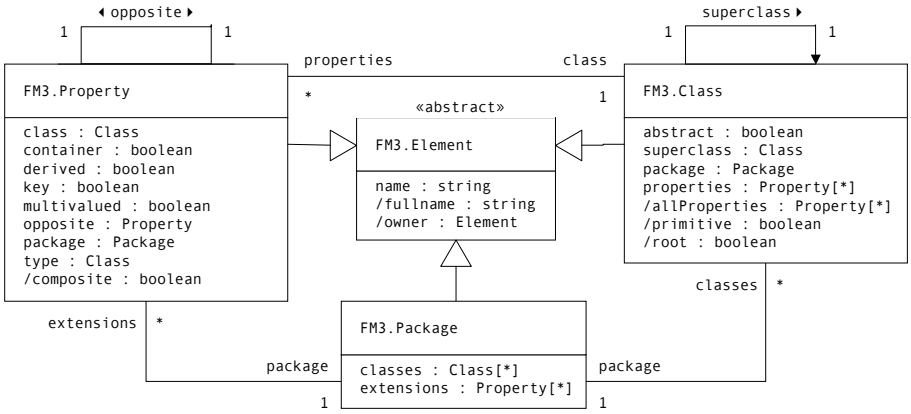

Fig. 2. Complete class diagram of FM3 meta-metamodel, including all properties and associations. Derived properties are marked with a leading slash character.

4 Meta-Metamodel —— Classes – Properties – Packages

All elements contained in a meta-repository must conform to FM3. These are all elements on the M2 and the M3 layer of a tower. Since the M3 layer is usually populated with elements that represent FM3, it conforms to itself.

C.A.R Hoare once said that "inside every large program there is a small program trying to get out," and in fact—FM3 is MOF 2.0 [8] reduced to the minimum. Meta, a precursor of Fame, has been used for many years as the kernel of Moose. Meta used MOF and later EMOF as its meta-metamodel. However, code reviews had shown that we only ever used a very small subset of EMOF, while many of the rarely used features had repeatedly been the cause of additional engineering effort.

Figure 2 presents the complete class diagram of FM3. There are three kinds of elements: packages, fame-classes, and properties. Packages contain classes and properties, classes contain properties. Packages can also directly contain properties which were not declared in the fame-class itself. In this case we say that those packages extend the fame-class and refer to the properties as *extensions*.

Extensions are one of the two additional FM3 features not present in EMOF. The other is unique keys. Both facilitate better metamodeling at runtime.

Extension properties are related to packaging and modularization of metamodels. Given package A and package B, package B can use extension properties to add properties to classes contained in A without tampering with the definition of A itself. This is useful for evolving applications. For example, for plugins or libraries that choose to extend the metamodel of an existing application.

Unique keys are related to the composition of associated elements. A fame-class must not have more than one keyed property. If an element with a keyed property k is contained in a container, the value of k must be unique with regard to the set of all values of k of all elements contained in the same container. This is useful for the improvement of the performance of runtime elements (discussed below in Section 7).

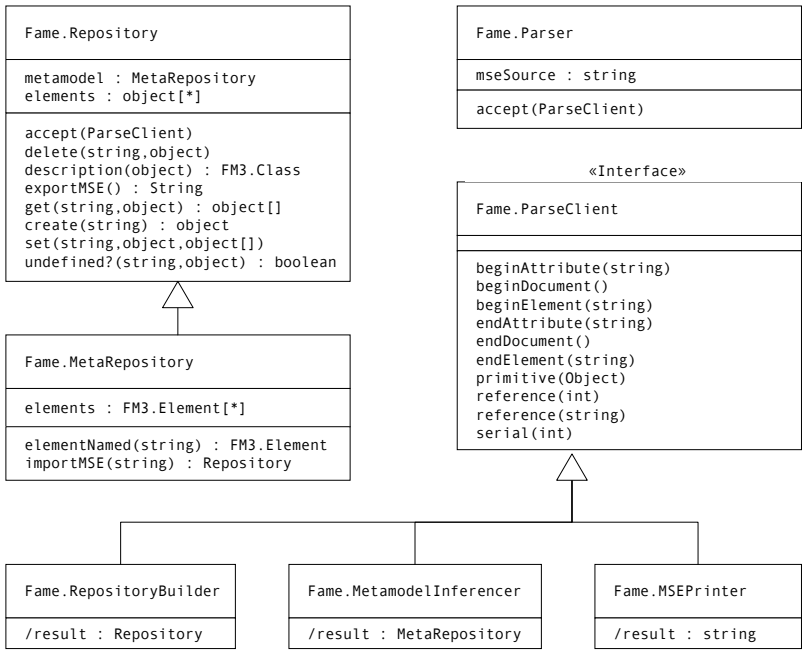

Fig. 3. Class diagram of the Fame API: (left) interfaces for runtime manipulation and elements and models, (right and bottom) interfaces for serialization and streaming.

The container can keep the composites in a host-dictionary rather than a host-collection, and thus provide access to composites by key in $O(1)$ rather than $O(n)$ time.

5 Fame API —– Create – Delete – Get – Set – Undefined

Since Fame is a polyglot library we have taken care to ensure that all implementations provide the same API. Fame offers functionality to

- manage one or more tower of models,
- manipulate elements and their meta-information,
- serialize and de-serialize (meta)models.

All elements (both objects and meta-information) are manipulated with the same API calls. This is possible since within a tower of models, objects and meta-information are "meta-described" in the same way. This allows for both models and metamodels to be serialized to the same exchange format.

Figure 3 presents the Fame API, from left to right/bottom: Repository and MetaRepository offer functionality to manipulate elements. Whereas Parser, ParseClient and its subclasses offer functionality to serialize and de-serialize complete (meta)models.

All elements have *attributes*. Each attribute has a name and optionally refers to another element. For each property of an element's fame-class, there must be a corre-

sponding attribute with the same name as the property. The value of an attribute is either set (that is, referring to another element) or empty and thus undefined[2].

Depending on host-language and scenario, attributes are either realized as instance variables or using separate host-classes. In particular, bi-directional relations are often realized as separate host-classes so that manipulation of one end automatically updates the opposite end.

The Fame API offers these five operations on elements:

- *create(name)* creates a new element that conforms to the specified fame-class. All attributes are initially undefined (unless specified otherwise by their type).
- *delete(slot, element)* resets the value of an attribute to undefined.
- *get(slot, element)* either returns the value of the attribute or fails if the value is undefined.
- *set(slot,element,value)* sets the value of a slot to the specified value.
- *undefined?(slot,element)* returns true if the slot is undefined or false otherwise.

This set of operations is similar to the five methods of RESTful web-programming [9]. This is not a coincidence, as many of the actions on business objects can be expressed in terms of common operations on the object graph of a running application. The same applies to the manipulation of meta-information. Fame does not offer dedicated operations for the meta-information. Rather, meta-information is manipulated with the same operations, the only difference being that we are operating another layer of the model-tower. The operation *description(element)* is used to navigate from an element to its meta-information.

Depending on the host-language and scenario, these operations of the Fame API are offered on the Repository host-class only (as indicated by Figure 3) or on native objects as well. The latter is not supported by host-languages where host-classes are closed for extension, which is the case for Java, but not C# and the others.

6 Serialization and Streaming —— Parse – Infer – Print

Elements of any layer are serializable to text stream and back using the MSE exchange format. MSE is a generic file format to exchange any kind of objects, regardless of the metamodel. Thus, FAME-compliant applications can exchange both data (*i.e.* models) and metamodels by the same means.

Serialization of (meta)models is based on streaming. Figure 3 presents the main collaborators of streaming to the right and the bottom. There are two kinds of collaborators: producers and consumers. Producers are host-classes that offers an *accept(ParseClient)* method. Consumers are host-classes that implement the ParseClient interface.

Fame offers two default producers:

Parser parses an MSE-compliant text stream; and
Repository iterates over all contained elements.

[2] Please note, that undefined and nil are not the same: according to the FM3 specification, nil is a predefined instance of Object, whereas undefined describes the value of an attribute. The value of an attribute that holds a reference to nil is thus not undefined.

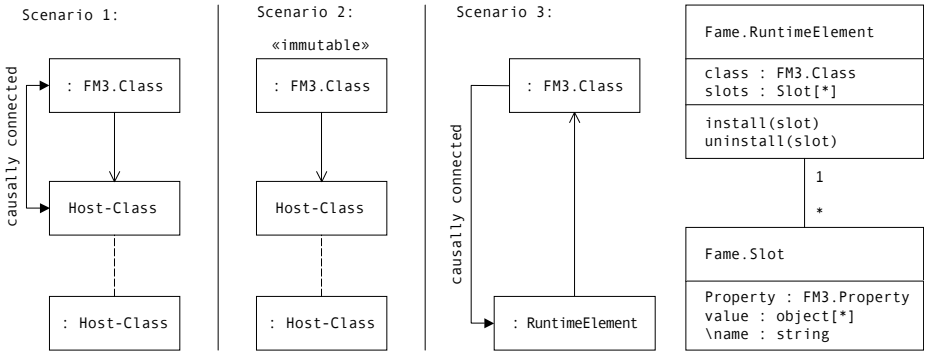

Fig. 4. Three scenarios of causal connection in Fame: (left), (center), (right).

It also offers default consumers:

RepositoryBuilder reads models given the stream of a model;
MetamodelInferencer infers the metamodel given the stream of a model; and
MSEPrinter serializes (meta)models to text streams.

The combination of inferencer and builder is able to import models for which the metamodel is not known or not available for some reason.

Models are exchanged between producers and consumers by means of a strict protocol. This communication is unidirectional. The producers fire a series of events to which the consumers listen. In case the protocol is not followed, the entire sequence is considered to be illegal. Each event corresponds to a method in the ParseClient interface. These methods must be called in following order (given as grammar rules):

$$
\begin{aligned}
\text{Sequence} &= \text{Document}\ ? \\
\text{Document} &= \mathit{beginDocument}\ \text{Element}\ *\ \mathit{endDocument} \\
\text{Element} &= \mathit{beginElement}\ \mathit{serial}\ ?\ \text{Attribute}\ *\ \mathit{endElement} \\
\text{Attribute} &= \mathit{beginAttribute}\ \text{Value}\ *\ \mathit{endAttribute} \\
\text{Value} &= \text{Element} \mid \mathit{reference} \mid \mathit{primitive}
\end{aligned}
$$

In fact, the above protocol corresponds to the grammar of the MSE exchange format[3] as specified in Appendix A. To avoid dependencies between streaming and meta-architecture, primitive parameters are only used in event signatures. Thus, serialized elements can be processed without setting up the entire meta-architecture first.

7 Causal Connection Between Fame-Classes and Host-Classes

Depending on host language and scenario, a causal connection between fame- and host-classes is maintained or not. Causal connection is required to update elements when

[3] The original meaning of MSE has been lost in the dust of history, however MSE is sometimes refered to as “Moose without objects” in reference to Fame’s origin as kernel of Moose.

their meta-information changes. There are three possible scenarios, as illustrated on Figure 4 from left to right:

- Each fame-class is associated with a corresponding host-class between which full causal connection is established. Changes to fame-classes are propagated to host-classes, and vice versa. The scenario is the default use case for languages that support hot swapping. Currently these are Smalltalk and Ruby, although it is also perfectly possible for Python.
- Each fame-class is associated with a corresponding host-class, but no causal connection is established. In fact, both classes are considered immutable and attempts to change fame-classes results in failure. Even though this scenario partly defies the purpose of Fame, it is default for Java, C# and Python.
- All model elements are instances of `RuntimeElement`, a dedicated host-class which emulates instances of fame-classes by using host-dictionaries. In this case fame-classes are not associated with host-classes, but rather directly referred to by the runtime elements. Causal connection between fame-classes and runtime elements is maintained. Changes to fame-classes are propagated and applied to all associated runtime elements. Currently this scenario is available for Smalltalk and Java.

Fame is a polyglot library and has been implemented (at varying stages of standard conformance) for many languages, each with its strengths and weaknesses. Even though all implementations offer the same core features, they differ in the degree to which causal connection between fame-classes and host-classes is maintained.

Due to missing hot swapping support by the host-language, the Java implementation offers the least degree of causal connection. The Smalltalk one offers the highest degree of causal connection.

In Smalltalk, FAME has been leveraged to the same level of abstraction and tool support as the host-language. For example, each manipulation that is applied to a metamodel is translated into a corresponding refactoring that is applied at runtime to the host-classes. Furthermore, both IDE and debugging tools have been extended to allow developers to inspect and manipulate the meta-information of any object at runtime.

Smalltalk development happens at runtime. You can think of its IDE as an advanced graphical REPL interface, and of editing source code as applying a sequence of runtime refactorings. It was only natural to extend these runtime development tools with support for metamodeling.

The degree of causal connection implies the degree to which a Fame implementation can bridge the gap between host-language and metamodeling at runtime. The above scenarios differ as follows

- Under Scenario 1 there is almost no gap between the host-language and metamodeling. Manipulating elements with host-language constructs or using the Fame API is equivalent, both have the same semantics.
- Under Scenario 3, RuntimeElement introduces a gap between host-language and metamodeling. All operations on runtime elements must use the Fame API.

For example, given element e in model m, there are two ways to set the value of a slot $zork$: either using host-language constructs `e.zork(value)`, or using the Fame API `m.set("zork",e,value)`. Scenario 3 supports the latter only.

Given a fully metamodel-driven application however, the difference between Scenario 1 on the one hand and Scenario 3 on the other hand is moot. Such an application will typically not require any code that bypasses the Fame API and can thus be realized under any of those two given scenarios of causal connection. Only Scenario 2 differs from the others as it is immutable. Any usage of the part of the Fame API which alters metamodels will result in failure.

A good example of a FAME-based application which is (almost) fully metamodel-driven is provided by Ducasse et al [6]. They present Moose, a research tool for software- and information visualization, that

- generates all UIs at runtime based on meta-information,
- exclusively uses the Fame API to interact with domain objects,
- exclusively uses the Fame API to interact with meta-information,
- exclusively uses Fame streaming to (de)serialize its data.

Moose is written in Smalltalk which employs Scenario 1. As long as Fame's API is never bypassed using host-language constructs, the same (or at least a similar) application could be written using Scenario 3 without loss of features or functionality. However, it remains open at which cost of runtime performance and development overhead.

8 Conclusion

FAME is a polyglot library for metamodeling at runtime. Fame attaches meta-information to the objects of a running application. The attached meta-infomration itself is described by another layer of meta-information, which is eventually described by itself. The API of Fame offers a common set of instructions to manipulate elements at any layer.

Special care is taken to integrate the meta-information as seamless as possible into the structures of the host-language. In particular, to establish causal connection between fame-classes and host-classes. As some hosts only offer limited reflection features, not all implementations of Fame offer the same degree of causal connection.

Due to missing hot swapping support by the host-language, of all FAME implementations, the Java one offers the least degree of causal connection. On the other hand, the Smalltalk one offers the highest degree of causal connection.

Fame supports three different scenarios of causal connection: 1) full causal connection, 2) no causal connection, and 3) emulated causal connection. All three are presented and discussed in the paper. We show that fully metamodel-driven applications can be written for both Scenario 1 and Scenario 3, as long as no host-language constructs are used to bypass Fame's metamodeling API.

Acknowledgments

We thank the anonymous reviewers for their corrections and interesting comments.

We gratefully acknowledge the financial support of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" and the Swiss National Science Foundation for the project "Analyzing, Capturing and Taming Software Change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

A Grammar of MSE Exchange Format

```
Root := Document ?
Document := OPEN ElementNode * CLOSE
ElementNode := OPEN NAME Serial ? AttributeNode * CLOSE
Serial := OPEN ID INTEGER CLOSE
AttributeNode := OPEN Name ValueNode * CLOSE
ValueNode := Primitive | Reference | ElementNode
Primitive := STRING | NUMBER | TRUE | FALSE | NIL
Reference := IntegerReference | NameReference
IntegerReference := OPEN REF INTEGER CLOSE
NameReference := OPEN REF NAME CLOSE

CLOSE := ")"
FALSE := "false"
ID := "id:"
INTEGER := digit +
NAME := letter ( letter | digit ) * ( "." letter ( letter | digit ) ) *
NUMBER := "-" ? digit + ( "." digit + ) ? ( "e" "-" ? digit + ) ?
OPEN := "("
REF := "ref:"
STRING := ( "'" [^'] * "'" ) +
TRUE := "true"
comment := "\"" [^"] * "\""
digit := [0-9]
letter := [a-zA-Z_]
whitespace := "\f" | "\n" | "\r" | "\s" | "\t"
```

All MSE text streams must use UTF-8 encoding.

References

1. Nierstrasz, O., Denker, M., Gîrba, T., Kuhn, A., Lienhard, A., Röthlisberger, D.: Self-aware, evolving eternal systems. Technical Report IAM-08-001, University of Berne, Institute of Applied Mathematics and Computer Sciences (2008)
2. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), New York NY, ACM Press (2005) 1–10 Invited paper.
3. Fabio Costa, Lucas Provensi, F.V.: Towards a more effective coupling of reflection and runtime metamodels for middleware. In: Workshop on Models at Runtime. (2006)
4. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Springer (2006) 171–185
5. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied metamodelling: A foundation for language driven development (2004)
6. Ducasse, S., Gîrba, T., Kuhn, A., Renggli, L.: Meta-environment and executable metalanguage using smalltalk: an experience report. Journal of Software and Systems Modeling (SOSYM) (2008) To appear.
7. Renggli, L., Ducasse, S., Kuhn, A.: Magritte — a meta-driven approach to empower developers and end users. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: Model Driven Engineering Languages and Systems. Volume 4735 of LNCS., Springer (September 2007) 106–120
8. Group, O.M.: Meta object facility (MOF) 2.0 core final adopted specification. Technical report, Object Management Group (2004)
9. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly (2007)